GPU Programming

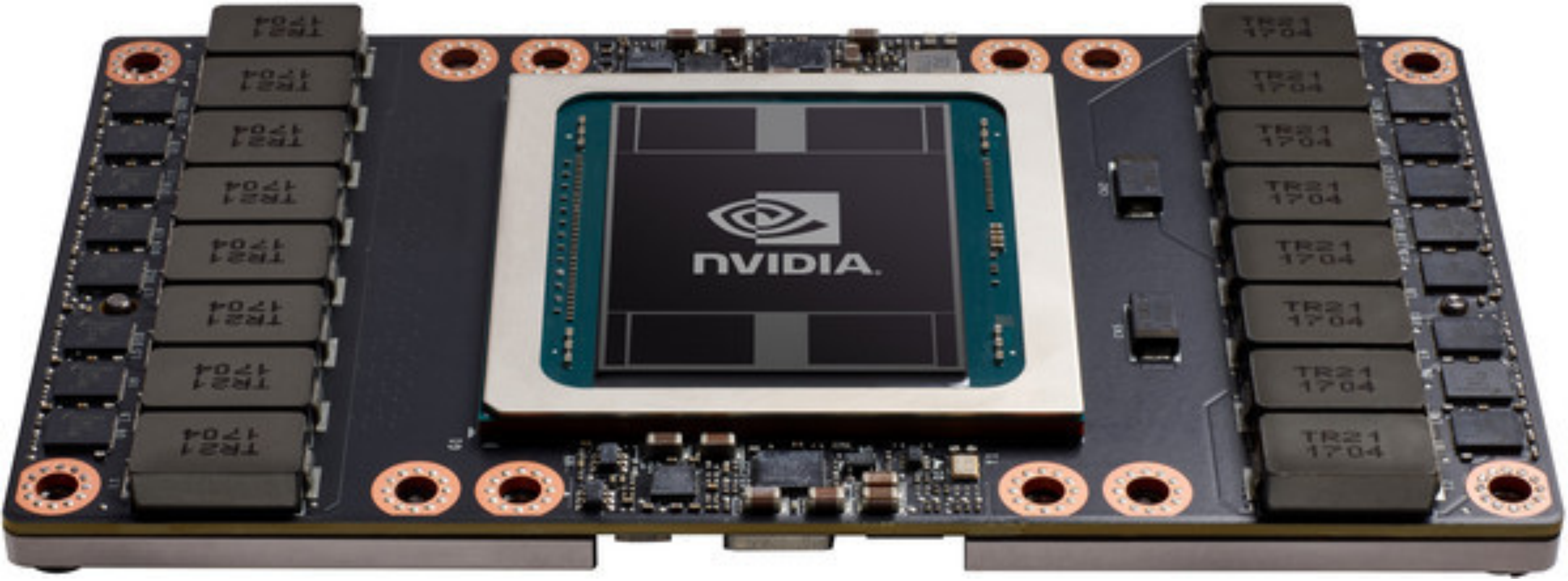
More details and related technologies

Franco Milicchio

https://fmilicchio.bitbucket.io

GPU Availability

- GPUs are readily available on many platforms
- PCs (obviously) and servers can be used, but also mobile architectures have GPUs
- Apple’s iOS has the *Metal Compute* framework, Android could have an OpenCL driver
- Intel itself did not stand by and released in 2012 the *Intel Xeon Phi* accelerator, after the failure of the Larrabee processor



NVidia Volta

Released in December 2017

Limits of GPUs

- As we've seen, we can have a 7× to 60×, or even 500× speedup compared to CPUs
- We have *a lot* of cores that *must* be fed with data
- However, the *bandwidth* is limited
- For instance, with an element-wise vector product, to feed the GPU we would need a 1 TB/s bandwidth
- We're not there yet (but we will be) and no latency or scheduling can hide this problem

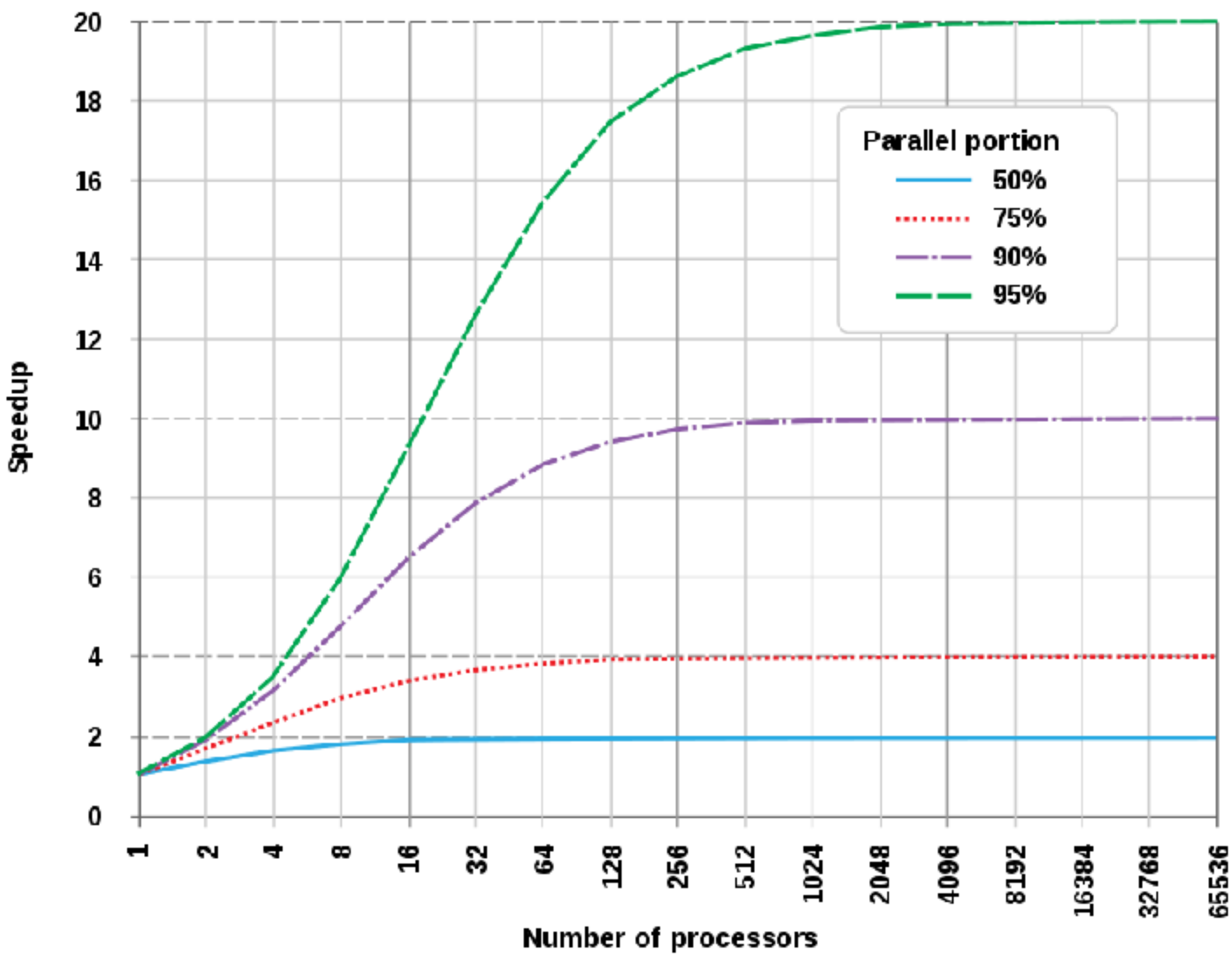
Vector Sum

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main(void)
{
    // ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    // ...
}
```

Nope!

Rule of thumb: do *more math* per data

Amdhal's Law


Parallel portion
50%
75%
90%
95%
Speedup
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of processors

Memory Model

Memory Management

- CUDA is an *explicit* tool for manycore programming
- Know that *there exists no such thing as a free lunch*
- Speed comes at a cost: you must manage thread invocation as we've seen
- Additionally, you *must* manage memory in the C way
- Allocating and deallocating memory is expensive, copying to and from the host is expensive: be careful

Memory Allocation

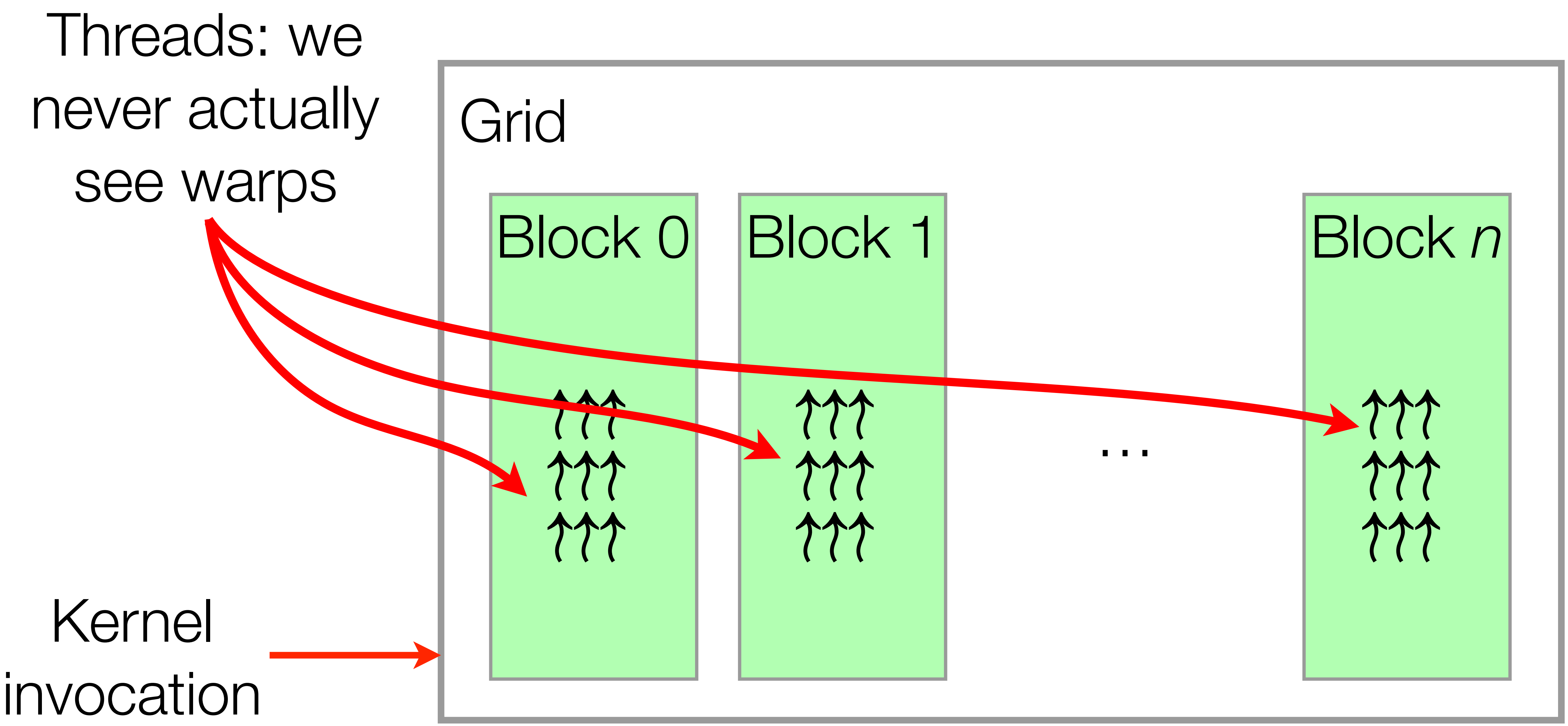
```
// Host code
int main(void)
{
    int N = /* ... */;
    size_t size = N * sizeof(float);

    float* h_A = (float*) malloc(size);
    float* h_B = (float*) malloc(size);

    // Initialize h_A and h_B on the CPU

    float* d_A;
    cudaMalloc((void**)&d_A, size);


    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // ...
```

This was on Windows

Memory Model

- Let’s recall that with CUDA we have a shared big memory on the GPU
- Then we allocate a chunk of memory where threads will find the input and output data
- Each *parallel invocation* of a *kernel* on the GPU is referred to as a *block*
- We have a *grid* of blocks then, *i.e.*, the set of all blocks
- Each block will have threads, actually scheduled in *warps*

Grids and Blocks



Grids and Blocks

```
addVector<<<1, N>>>(x, y, z);
```

One invocation
with multiple threads

```
addVector<<<N, 1>>>(x, y, z);
```

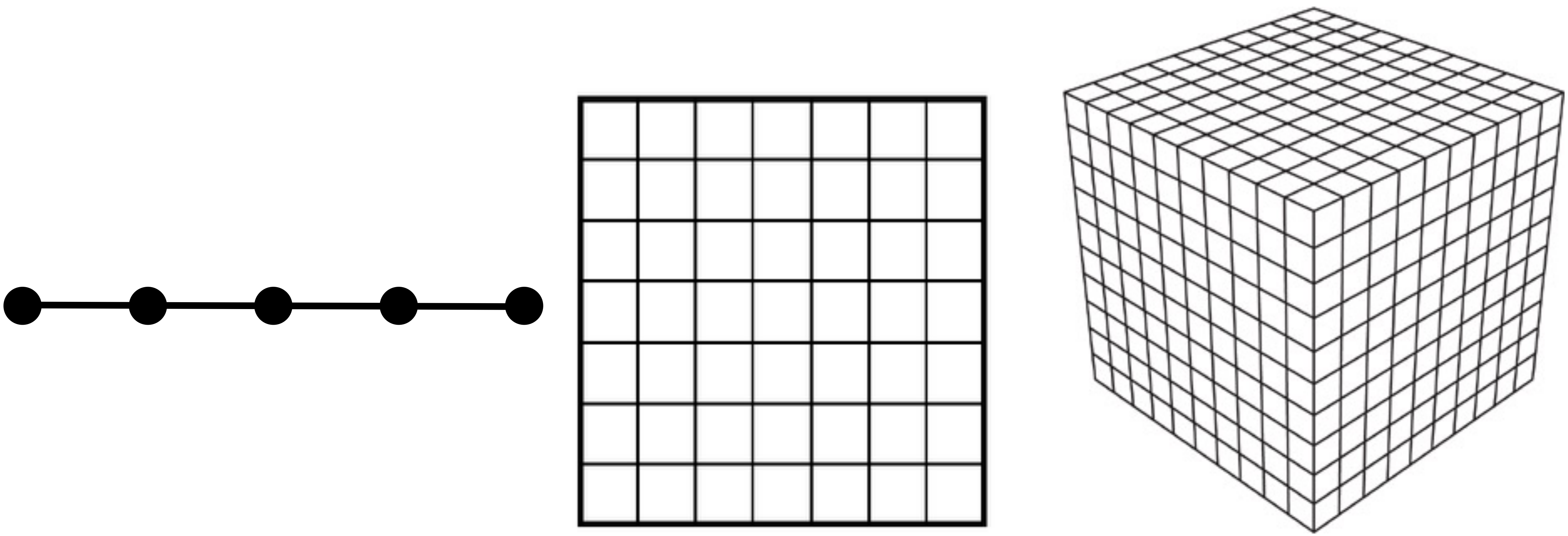
Multiple invocations
with one thread

Beware: the number of threads that a *block* can handle is hardware-limited, a good limit is 1024 threads per block

Kernel Invocations

- The two parameters passed with the triple angular parentheses are, as said, necessary
- In reality, there are four, but we won't see all of them here
- The first is the the number of *blocks*
- The second is the number of *threads* inside the block
- We have seen numbers, but actually they are *not* numbers

Memory Mapping



Memory Maps

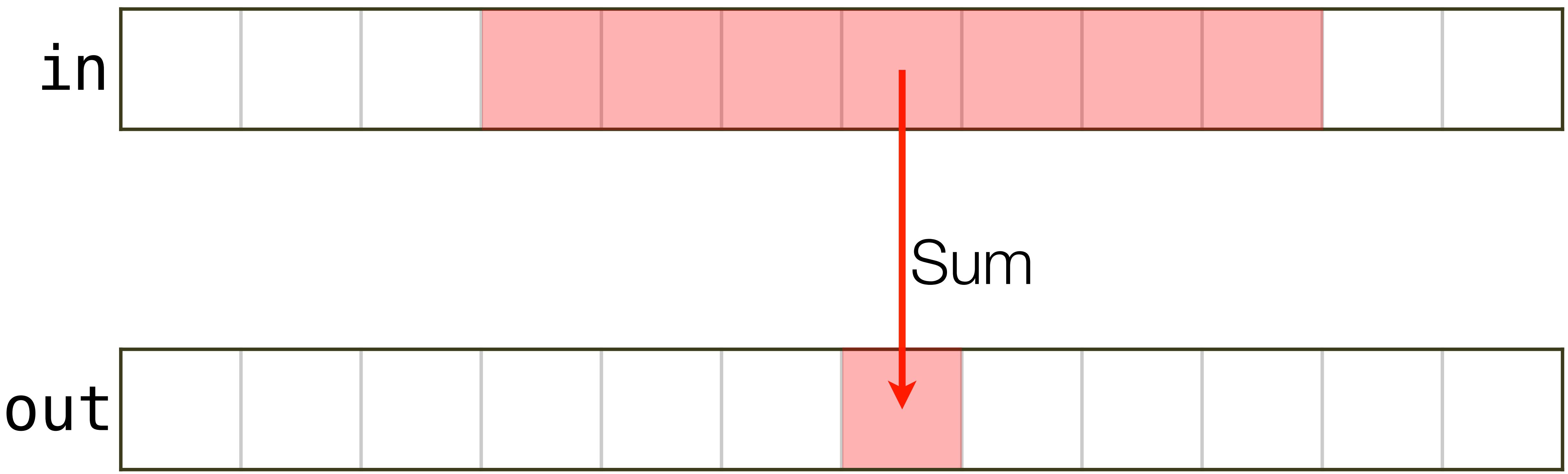
- Each parameter is of type `dim3`, and can have up to three dimensions (obviously)
- What dimension is needed depends on your problem
- This influences how you program, not actually how the GPU accesses memory (which is, in fact, a linear space)
- A block has index `blockIdx` (with x, y, and z), and `threadIdx` thread index (again with x, y, and z)
- Remember that a thread index refers to the index *inside the block* and not a global one

Threads

Cooperation

- Let's consider a simple example in order to answer the following question
- Why would we want threads when we already have blocks?
- In this example, we will use a one-dimensional stencil
- We have a vector, and we want as output a vector of the same size
- The *i*-th element shall contain the sum of the elements at distance 3 from *i*

Stencil

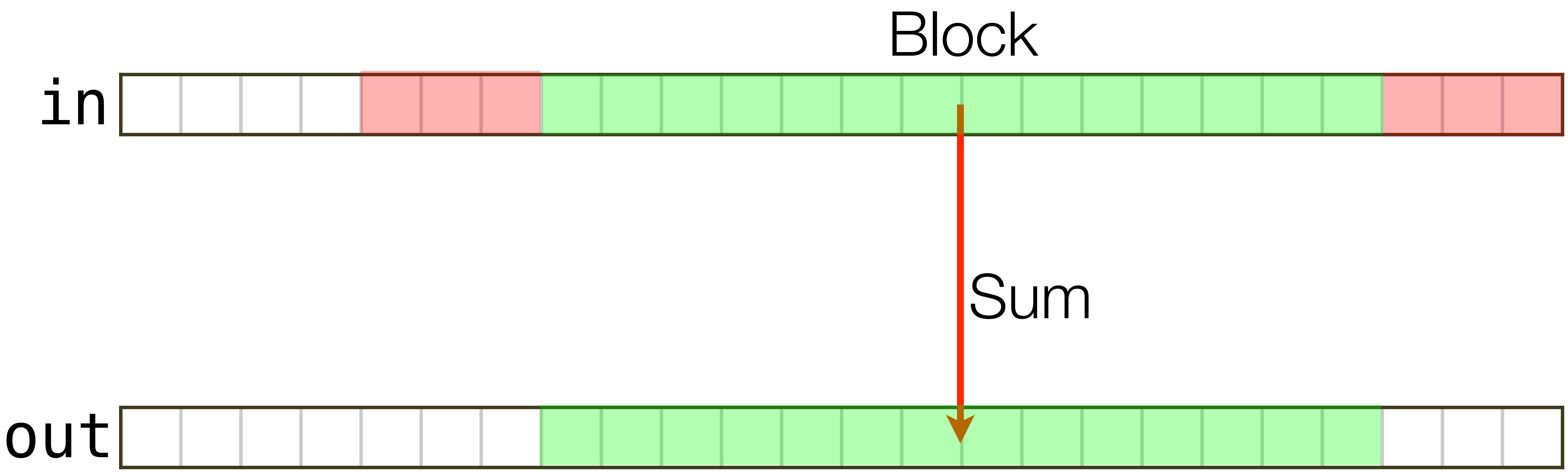


How many memory reads are there?

Optimizing Memory Transfers

- Each thread will process *one* output item, but each item in the input vector is read *seven* times
- This is extremely slow, and we should use a *shared* memory approach to minimize memory reads
- In fact, threads in a block can access a very fast shared (private) memory, visible only inside the block
- Then we could just read **`blockDim.x`** items to the shared memory, compute, and write back **`blockDim.x`** items

Shared Memory



Let's see the code

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out)
{
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gi = threadIdx.x + blockIdx.x * blockDim.x;
    int li = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[li] = in[gi];

    if (threadIdx.x < RADIUS)
    {
        temp[li - RADIUS]     = in[gi - RADIUS];
        temp[li + BLOCK_SIZE] = in[gi + BLOCK_SIZE];
    }

//...
```

Stencil Kernel

```
//...

    // Apply the stencil
    int result = 0;

    for (int o = -RADIUS ; o <= RADIUS ; o++)
        result += temp[li + o];

    // Store the result
    out[gi] = result;
}
```

This has a nasty error

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out)
{
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gi = threadIdx.x + blockIdx.x * blockDim.x;
    int li = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[li] = in[gi];

    if (threadIdx.x < RADIUS)
    {
        temp[li - RADIUS]     = in[gi - RADIUS];
        temp[li + BLOCK_SIZE] = in[gi + BLOCK_SIZE];
    }
```

Immagine thread 15 reads the last items while thread 0 hasn't fetched them: what would happen?

Data Races

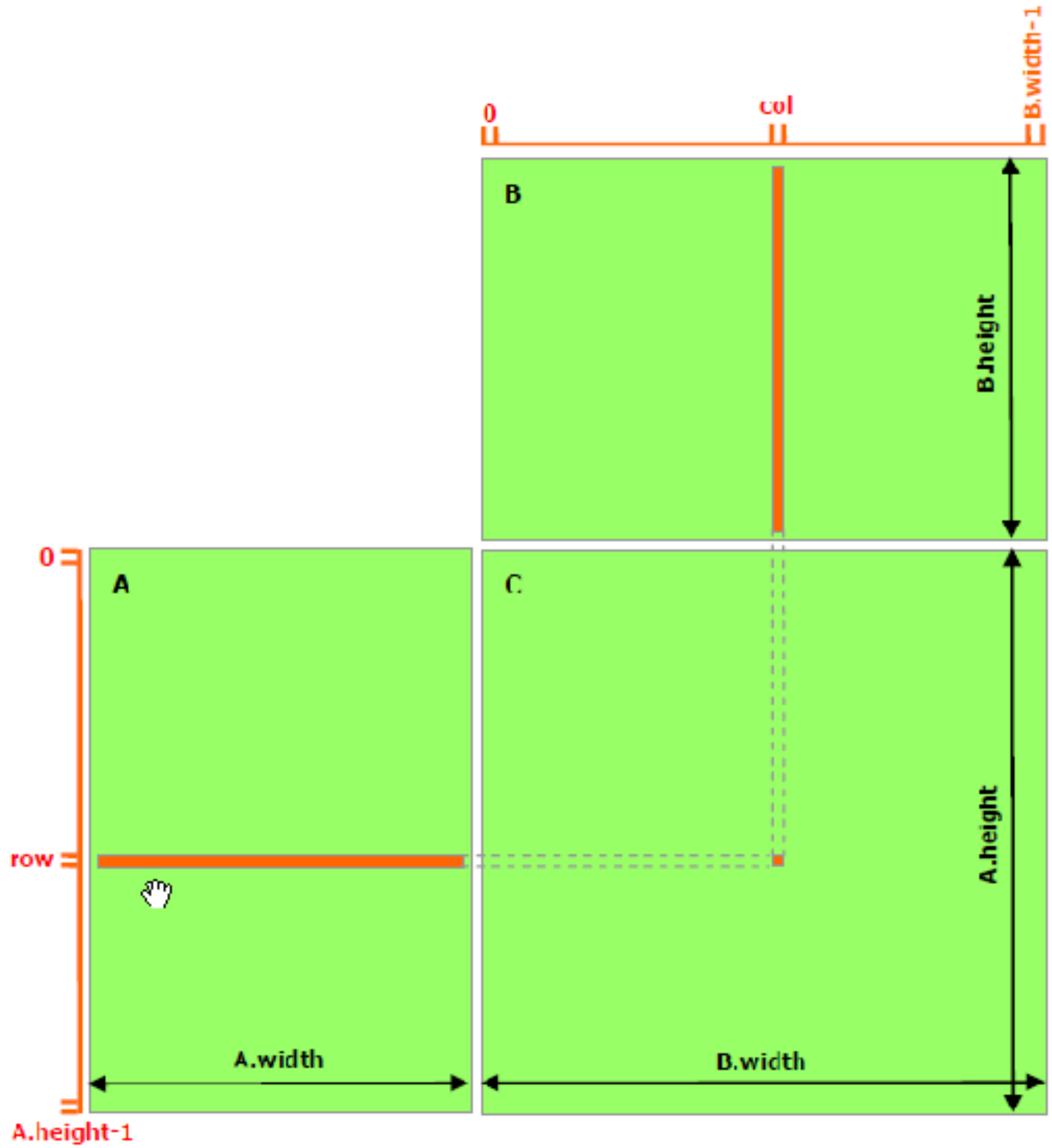
```
__global__ void stencil_1d(int *in, int *out)
{
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gi = threadIdx.x + blockIdx.x * blockDim.x;
    int li = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[li] = in[gi];

    if (threadIdx.x < RADIUS)
    {
        temp[li - RADIUS]     = in[gi - RADIUS];
        temp[li + BLOCK_SIZE] = in[gi + BLOCK_SIZE];
    }

    // Barrier
    __synchthreads();
```

Matrix Multiplication

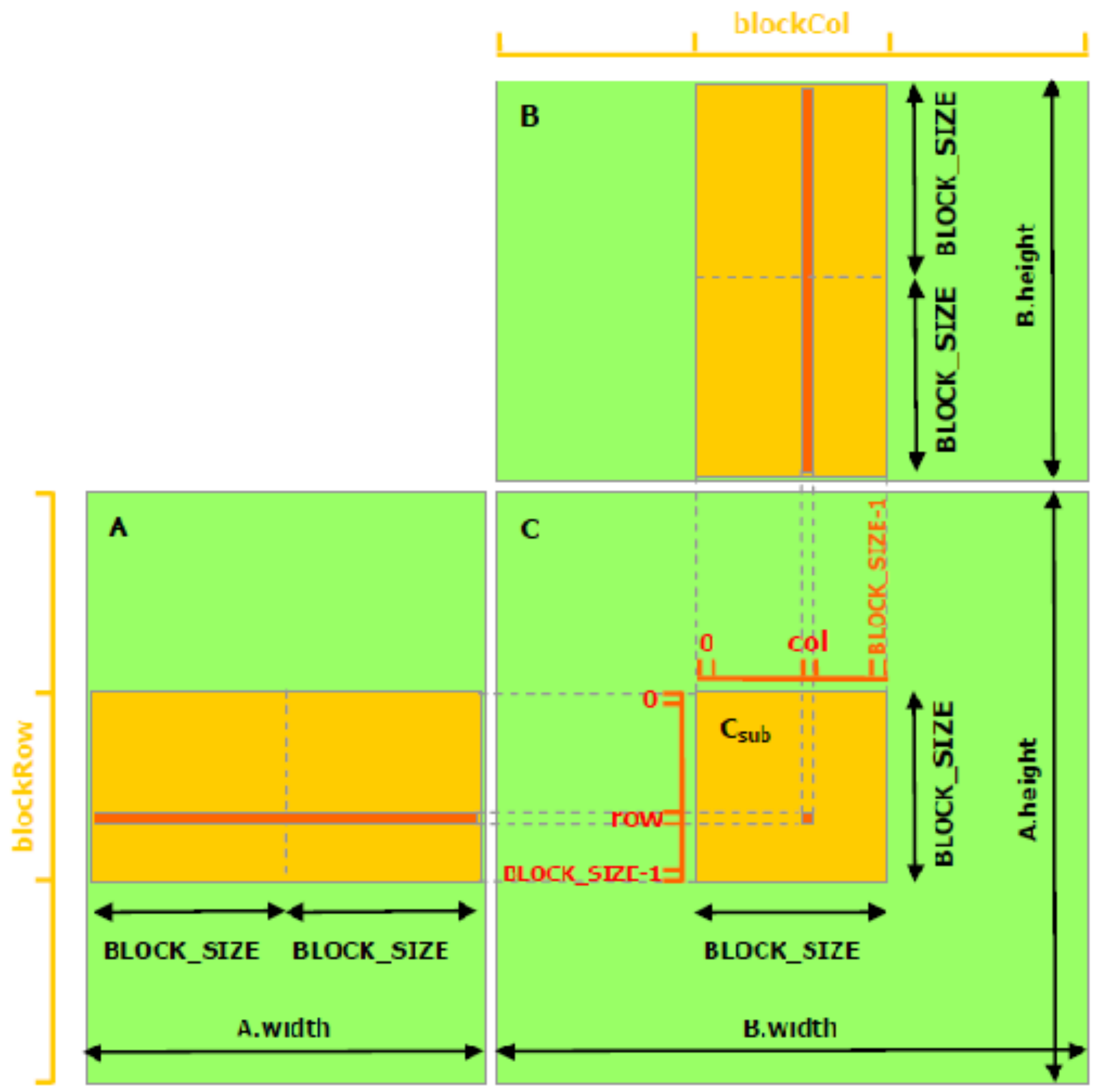
Matrix Multiplication

- Matrix multiplication is the basis of many applications
- Physical simulations, deep learning, optimizations, eigenproblems, all use matrix multiplications
- Without any shared memory, we will have sub-optimal speedups as we've seen with simple vectors
- Then, let's use a shared memory approach (you can find the code on NVidia's site)

Without Shared Memory



With Shared Memory



Matrix Multiplication

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct
{
   int    width, height, stride;
   float* elements;
} Matrix;

__device__ float GetElement(const Matrix A, int row, int col)
{
   return A.elements[row * A.stride + col];
}

__device__ void SetElement(Matrix A, int row, int col, float  value)
{
   A.elements[row * A.stride + col] = value;
}
```

Matrix Multiplication

```
// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub,
// located col sub-matrices to the right and row
// sub-matrices down from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A,
                               int row, int col)
{
   Matrix Asub;
   Asub.width  = BLOCK_SIZE;
   Asub.height = BLOCK_SIZE;
   Asub.stride = A.stride;

   Asub.elements =
      &A.elements[A.stride * BLOCK_SIZE * row +
                  BLOCK_SIZE * col];
   return Asub;
}
```

Matrix Multiplication

```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of matrix multiplication kernel
__global__ void MatMulKernel(const Matrix,
                            const Matrix, Matrix);
```

Matrix Multiplication

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
   // Load A and B to device memory
   Matrix d_A;
   d_A.width   = d_A.stride = A.width;
   d_A.height  = A.height;
   size_t size = A.width * A.height * sizeof(float);

   cudaMalloc((void**)&d_A.elements, size);
   cudaMemcpy(d_A.elements, A.elements, size,
              cudaMemcpyHostToDevice);

   Matrix d_B;
   d_B.width  = d_B.stride = B.width;
   d_B.height = B.height;
   size       = B.width * B.height * sizeof(float);

   cudaMalloc((void**)&d_B.elements, size);
   cudaMemcpy(d_B.elements, B.elements, size,
              cudaMemcpyHostToDevice);
```

Matrix Multiplication

```
    // Allocate C in device memory
    Matrix d_C;
    d_C.width  = d_C.stride = C.width;
    d_C.height = C.height;

    size       = C.width * C.height * sizeof(float);

    cudaMalloc((void**)&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);

    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Matrix Multiplication

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B,
                             Matrix C)
{
   // Block row and column
   int blockRow = blockIdx.y;
   int blockCol = blockIdx.x;

   // Each thread block computes one sub-matrix Csub of C
   Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

   // Each thread computes one element of Csub
   // by accumulating results into Cvalue
   float Cvalue = 0;

   // Thread row and column within Csub
   int row = threadIdx.y;
   int col = threadIdx.x;
```

Matrix Multiplication

```
// Loop over all the sub-matrices of A and B that are
// required to compute Csub
// Multiply each pair of sub-matrices together
// and accumulate the results
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m)
{
   // Get sub-matrix Asub of A
   Matrix Asub = GetSubMatrix(A, blockRow, m);

   // Get sub-matrix Bsub of B
   Matrix Bsub = GetSubMatrix(B, m, blockCol);

   // Shared memory used to store Asub and Bsub respectively
   __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

   // Load Asub and Bsub from device memory to shared memory
   // Each thread loads one element of each sub-matrix
   As[row][col] = GetElement(Asub, row, col);
   Bs[row][col] = GetElement(Bsub, row, col);
```

Matrix Multiplication

```
        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}
```